Building a REST API with FastAPI

1. What Are We Building?

In the previous FastAPI fundamentals lesson, we learned how FastAPI works.

Now we will build an actual **REST API**.

Our example will be a small **learning resource API** similar to the backend that could power a platform like haas.dev.

The API will support:

```
GET     /api/resources
GET     /api/resources/{id}
POST    /api/resources
PUT     /api/resources/{id}
PATCH   /api/resources/{id}
DELETE  /api/resources/{id}
```

These endpoints allow a client to:

- Read resources
- Create resources
- Update resources
- Partially update resources
- Delete resources
- Filter resources
- Validate incoming data
- Return appropriate HTTP status codes

FastAPI maps HTTP methods and URL paths to Python path operation functions, while Pydantic models can define and validate request and response data.

---

2. What Makes an API RESTful?

REST is an architectural style for designing networked applications.

A REST API generally organizes its interface around **resources**.

For our application:

```
Resource
```

is the main resource.

Instead of creating action-heavy URLs such as:

```
/getAllResources
/createResource
/deleteResource
```

we use:

```
/resources
```

and let the HTTP method communicate the operation.

```
GET    /resources
POST   /resources
PUT    /resources/1
PATCH  /resources/1
DELETE /resources/1
```

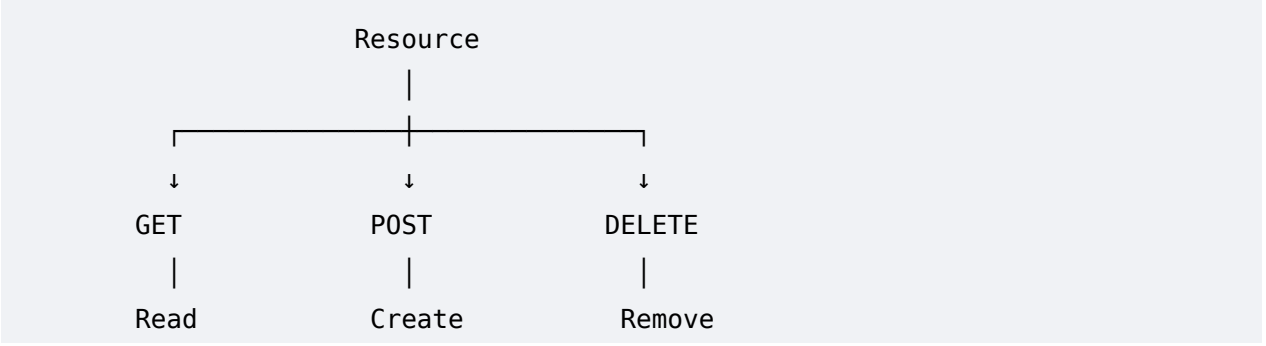
The URL identifies the resource.

The HTTP method describes the intended operation.

---

3. REST Mental Model

Think about the API like this:



For an individual resource:

```
GET /resources/5
```

means:

```
Give me resource 5.
```

While:

```
DELETE /resources/5
```

means:

```
Remove resource 5.
```

The URL remains related to the same resource.

The HTTP method changes what we want to do.

---

4. REST API Architecture

A real application might look like:

```
Frontend / Mobile App
          ↓
      HTTP Request
          ↓
     FastAPI Router
          ↓
       Service
          ↓
    Repository / ORM
          ↓
       Database
          ↓
    Repository / ORM
          ↓
       Service
          ↓
     FastAPI Router
          ↓
      JSON Response
```

For a small project, we can simplify this:

```
Client
  ↓
FastAPI
  ↓
Python logic
  ↓
In-memory data
  ↓
JSON
```

Then later replace the in-memory data with a database.

---

5. Project Setup

Create a project:

```
fastapi-rest-api/
```

Create a virtual environment:

```
python -m venv .venv
```

Activate it.

Windows

```
.venv\Scripts\activate
```

macOS/Linux

```
source .venv/bin/activate
```

Install FastAPI:

```
pip install "fastapi[standard]"
```

Save dependencies:

```
pip freeze > requirements.txt
```

A current FastAPI development workflow can use the `fastapi dev` command to run the development server.

---

6. Project Structure

Start simple:

```
fastapi-rest-api/
│
├── .venv/
├── main.py
└── requirements.txt
```

As the application grows, we will eventually move toward:

```
fastapi-rest-api/
│
├── app/
│   ├── main.py
│   ├── routers/
│   ├── schemas/
```

```
│   ├── services/
│   ├── models/
│   └── dependencies.py
│
├── tests/
├── requirements.txt
└── .env
```

Do not create a complicated architecture before you actually need it.

---

7. Create the FastAPI Application

Create `main.py`:

```
from fastapi import FastAPI

app = FastAPI(
    title="Resource API",
    description="A REST API for learning resources",
    version="1.0.0"
)


@app.get("/")
def root():
    return {
        "message": "Resource API is running"
    }
```

Run:

```
fastapi dev
```

Open:

```
http://127.0.0.1:8000/
```

You should receive:

```
{
    "message": "Resource API is running"
}
```

---

8. Automatic Documentation

FastAPI automatically generates interactive API documentation.

Open:

```
http://127.0.0.1:8000/docs
```

You can:

- View endpoints
- See request parameters
- See request bodies
- See response schemas
- Execute requests
- Inspect responses

FastAPI also provides ReDoc at:

```
http://127.0.0.1:8000/redoc
```

FastAPI generates an OpenAPI schema for the application, which powers these documentation interfaces.

---

9. Define Our Resource

Our learning resource will contain:

```
id
title
description
category
level
```

For example:

```
{
    "id": 1,
    "title": "Python Basics",
    "description": "Learn the fundamentals of Python.",
    "category": "Python",
    "level": "Beginner"
}
```

---

10. Create a Pydantic Model

Import:

```
from pydantic import BaseModel
```

Create:

```
class Resource(BaseModel):
    id: int
    title: str
    description: str
    category: str
    level: str
```

This model describes the shape of a resource.

---

11. Create Input and Output Models

A client should not normally provide its own database ID when creating a resource.

Therefore, separate the models.

```
class ResourceCreate(BaseModel):
    title: str
    description: str
    category: str
    level: str


class ResourceResponse(BaseModel):
    id: int
    title: str
    description: str
    category: str
    level: str
```

Now:

```
Client sends
    ↓
ResourceCreate

Server returns
    ↓
ResourceResponse
```

This distinction becomes increasingly important in real applications.

---

12. Our Temporary Data Store

For learning, use a Python list:

```
resources = [
    {
        "id": 1,
        "title": "Python Basics",
        "description": "Learn Python fundamentals.",
        "category": "Python",
        "level": "Beginner"
    },
    {
        "id": 2,
        "title": "FastAPI Fundamentals",
        "description": "Learn the basics of FastAPI.",
        "category": "Python",
        "level": "Intermediate"
    }
]
```

Important:

This is **not a real database**.

If the application restarts, the data returns to its original state.

We are using it only to understand REST API design.

---

13. GET All Resources

Create:

```
@app.get(
    "/api/resources",
    response_model=list[ResourceResponse]
)
def get_resources():
    return resources
```

Request:

```
GET /api/resources
```

Response:

```
[
    {
        "id": 1,
        "title": "Python Basics",
        "description": "Learn Python fundamentals.",
        "category": "Python",
        "level": "Beginner"
    },
    {
        "id": 2,
        "title": "FastAPI Fundamentals",
        "description": "Learn the basics of FastAPI.",
        "category": "Python",
        "level": "Intermediate"
    }
]
```

The response model tells FastAPI what shape the returned data should have and is also used for validation, serialization, documentation, and filtering the output.

---

14. GET One Resource

Add a path parameter:

```
from fastapi import HTTPException


@app.get(
    "/api/resources/{resource_id}",
    response_model=ResourceResponse
)
def get_resource(resource_id: int):

    for resource in resources:
        if resource["id"] == resource_id:
            return resource

    raise HTTPException(
        status_code=404,
        detail="Resource not found"
    )
```

Request:

```
GET /api/resources/1
```

Response:

```
{
    "id": 1,
    "title": "Python Basics",
    "description": "Learn Python fundamentals.",
    "category": "Python",
    "level": "Beginner"
}
```

If resource `99` does not exist:

```
GET /api/resources/99
```

returns a `404`.

---

15. Why Use `HTTPException`?

Instead of:

```
return {
    "error": "Not found"
}
```

we can explicitly tell FastAPI:

```
raise HTTPException(
    status_code=404,
    detail="Resource not found"
)
```

The client receives an HTTP error response.

This matters because the client can inspect the status code:

```
200 → resource found
404 → resource does not exist
```

---

16. POST: Creating a Resource

Now we need:

```
POST /api/resources
```

The client sends:

```
{
    "title": "Building REST APIs",
    "description": "Learn REST API development.",
    "category": "Backend",
    "level": "Intermediate"
}
```

Create the endpoint:

```
from fastapi import status


@app.post(
    "/api/resources",
    response_model=ResourceResponse,
    status_code=status.HTTP_201_CREATED
)
def create_resource(resource: ResourceCreate):

    new_resource = {
        "id": len(resources) + 1,
        **resource.model_dump()
    }

    resources.append(new_resource)

    return new_resource
```

FastAPI reads the JSON request body into the Pydantic model and validates it before the function executes.

A successful creation should normally use:

```
201 Created
```

FastAPI allows the status code to be declared directly on the path operation and includes it in the generated OpenAPI schema.

---

17. Request Body Flow

When the client sends:

```
{
    "title": "Building REST APIs",
    "description": "Learn REST API development.",
    "category": "Backend",
    "level": "Intermediate"
}
```

the flow is:

```
JSON
 ↓
FastAPI
 ↓
Pydantic validation
 ↓
ResourceCreate
 ↓
Python function
 ↓
Create resource
 ↓
Response model
 ↓
JSON
```

This is one of the most important FastAPI patterns.

---

18. What If Required Data Is Missing?

Suppose the client sends:

```
{
    "title": "Building REST APIs"
}
```

But our model requires:

```
class ResourceCreate(BaseModel):
    title: str
    description: str
    category: str
    level: str
```

The request fails validation.

The application does not need to manually write:

```
if "description" not in data:
```

for every field.

The model describes the contract.

---

19. PUT: Updating a Resource

PUT can represent a complete update of a resource.

Create:

```
@app.put(
    "/api/resources/{resource_id}",
    response_model=ResourceResponse
)
def update_resource(
    resource_id: int,
    resource: ResourceCreate
):

    for existing in resources:
        if existing["id"] == resource_id:

            existing.update(
                resource.model_dump()
            )

            return existing

    raise HTTPException(
        status_code=404,
        detail="Resource not found"
    )
```

Request:

```
PUT /api/resources/1
```

Body:

```
{
    "title": "Advanced Python",
    "description": "Deep dive into Python.",
```

```
    "category": "Python",
    "level": "Advanced"
}
```

The resource is replaced with the supplied representation.

---

20. PATCH: Partial Updates

PATCH is useful when the client wants to modify only certain fields.

For example:

```
PATCH /api/resources/1
```

Body:

```
{
    "level": "Advanced"
}
```

We need a model where all fields are optional:

```
class ResourceUpdate(BaseModel):
    title: str | None = None
    description: str | None = None
    category: str | None = None
    level: str | None = None
```

Then:

```
@app.patch(
    "/api/resources/{resource_id}",
    response_model=ResourceResponse
)
def patch_resource(
    resource_id: int,
    resource: ResourceUpdate
):

    for existing in resources:

        if existing["id"] == resource_id:

            update_data = resource.model_dump(
                exclude_unset=True
            )
```

```
            existing.update(update_data)

            return existing

    raise HTTPException(
        status_code=404,
        detail="Resource not found"
    )
```

The important part is:

```
exclude_unset=True
```

This allows us to distinguish:

```
Field not provided
```

from:

```
Field explicitly provided
```

---

21. PUT vs PATCH

Method	**Typical meaning**
PUT	Replace/update the resource representation
PATCH	Partially modify the resource

Example:

PUT

```
{
    "title": "Python",
    "description": "Learn Python",
    "category": "Programming",
    "level": "Beginner"
}
```

PATCH

```
{
    "level": "Intermediate"
}
```

The exact semantics should be documented by your API.

---

22. DELETE

Create:

```
@app.delete(
    "/api/resources/{resource_id}",
    status_code=status.HTTP_204_NO_CONTENT
)
def delete_resource(resource_id: int):

    for index, resource in enumerate(resources):

        if resource["id"] == resource_id:
            resources.pop(index)
            return

    raise HTTPException(
        status_code=404,
        detail="Resource not found"
    )
```

Request:

```
DELETE /api/resources/2
```

Successful response:

```
204 No Content
```

A `204` response should not contain a response body.

---

23. Complete CRUD Flow

Our REST API now supports:

```
CREATE
POST /api/resources
       ↓
201 Created
```

```
READ ALL
GET /api/resources
       ↓
200 OK
```

```
READ ONE
GET /api/resources/1
       ↓
200 OK
```

```
UPDATE
PUT /api/resources/1
       ↓
200 OK
```

```
PARTIAL UPDATE
PATCH /api/resources/1
       ↓
200 OK
```

```
DELETE
DELETE /api/resources/1
       ↓
204 No Content
```

This is the core CRUD structure of many REST APIs.

---

24. Add Filtering

Our API can support:

```
GET /api/resources?category=Python
```

Add:

```
@app.get(
    "/api/resources",
    response_model=list[ResourceResponse]
)
def get_resources(
    category: str | None = None
):

    if category:
        return [
            resource
            for resource in resources
            if resource["category"].lower()
            == category.lower()
        ]

    return resources
```

Now:

```
/api/resources
```

returns all resources.

While:

```
/api/resources?category=Python
```

returns only Python resources.

---

25. Add Level Filtering

We can add another query parameter:

```
@app.get(
    "/api/resources",
    response_model=list[ResourceResponse]
)
def get_resources(
    category: str | None = None,
    level: str | None = None
):

    result = resources
```

```
    if category:
        result = [
            resource
            for resource in result
            if resource["category"].lower()
            == category.lower()
        ]

    if level:
        result = [
            resource
            for resource in result
            if resource["level"].lower()
            == level.lower()
        ]

    return result
```

Now we can request:

```
/api/resources?category=Python&level=Beginner
```

---

26. Add Search

A search parameter:

```
/api/resources?search=flask
```

could search titles and descriptions.

```
@app.get(
    "/api/resources",
    response_model=list[ResourceResponse]
)
def get_resources(
    category: str | None = None,
    level: str | None = None,
    search: str | None = None
):

    result = resources

    if category:
        result = [
```

```
            resource
            for resource in result
            if resource["category"].lower()
            == category.lower()
        ]

    if level:
        result = [
            resource
            for resource in result
            if resource["level"].lower()
            == level.lower()
        ]

    if search:
        search = search.lower()

        result = [
            resource
            for resource in result
            if search in resource["title"].lower()
            or search in resource["description"].lower()
        ]

    return result
```

Now:

```
/api/resources?search=flask
```

can return resources related to Flask.

---

27. Pagination

Imagine the API eventually contains:

```
100,000 resources
```

Returning all of them at once is inefficient.

Instead:

```
GET /api/resources?skip=0&limit=20
```

We can implement:

```
@app.get(
    "/api/resources",
    response_model=list[ResourceResponse]
)
def get_resources(
    skip: int = 0,
    limit: int = 20
):
    return resources[skip:skip + limit]
```

Then:

```
skip=0&limit=20
```

means:

```
Resources 1–20
```

while:

```
skip=20&limit=20
```

means:

```
Resources 21–40
```

For production APIs, pagination should normally be backed by the database rather than slicing a large in-memory list.

---

28. Why Pagination Matters

Without pagination:

```
Client
  ↓
GET /resources
  ↓
500,000 records
  ↓
Huge response
  ↓
Slow network
  ↓
More memory
```

With pagination:

```
Client
  ↓
GET /resources?limit=20
  ↓
20 records
  ↓
Smaller response
```

Pagination becomes especially important for mobile clients and large datasets.

---

29. Add Metadata to Pagination

A more useful response could be:

```
{
    "data": [
        {
            "id": 1,
            "title": "Python Basics"
        }
    ],
    "page": 1,
    "limit": 20,
    "total": 150
}
```

Create models:

```
class ResourcePage(BaseModel):
    data: list[ResourceResponse]
    page: int
    limit: int
    total: int
```

Then return:

```
{
    "data": results,
    "page": page,
    "limit": limit,
    "total": len(resources)
}
```

This gives clients enough information to build pagination controls.

---

30. Consistent Error Responses

A basic error:

```
{
    "detail": "Resource not found"
}
```

is useful.

For a larger API, you may want a consistent structure:

```
{
    "error": {
        "code": "RESOURCE_NOT_FOUND",
        "message": "Resource not found"
    }
}
```

For example:

```
RESOURCE_NOT_FOUND
INVALID_RESOURCE
UNAUTHORIZED
FORBIDDEN
VALIDATION_ERROR
```

The important thing is consistency.

---

31. HTTP Status Codes

Our API can use:

Situation	Status
Successful GET	200
Resource created	201

Successful update	200
Successful delete with no body	204
Invalid request	400
Authentication required	401
Permission denied	403
Resource not found	404
Conflict	409
Validation failure	422
Server failure	500

FastAPI allows status codes to be configured directly on path operations and documents them through OpenAPI.

---

32. Response Models Are an API Contract

Suppose our internal resource contains:

```
{
    "id": 1,
    "title": "Python",
    "category": "Python",
```

```
    "internal_note": "Admin only"
}
```

But our response model contains:

```
class ResourceResponse(BaseModel):
    id: int
    title: str
    category: str
```

The client should receive:

```
{
    "id": 1,
    "title": "Python",
    "category": "Python"
}
```

not:

```
{
    "id": 1,
    "title": "Python",
    "category": "Python",
    "internal_note": "Admin only"
}
```

FastAPI uses response models for validation, serialization, OpenAPI documentation, and filtering the returned data to the declared shape.

This is both an API-design and security concern.

---

33. API Versioning

As an API evolves, clients may depend on its existing behavior.

You can version your endpoints:

```
/api/v1/resources
```

Later:

```
/api/v2/resources
```

For example:

```
@app.get(
    "/api/v1/resources",
    response_model=list[ResourceResponse]
)
def get_resources_v1():
    return resources
```

Versioning becomes useful when making breaking changes.

---

34. API Tags

Organize endpoints in the documentation:

```
@app.get(
    "/api/resources",
    tags=["Resources"]
)
def get_resources():
    return resources
```

You could have:

```
Resources
Users
Authentication
Categories
Admin
```

FastAPI supports tags and other metadata directly on path operations, and this information becomes part of the generated OpenAPI documentation.

---

35. Separate Routers

A larger API should not keep every route inside `main.py`.

Create:

```
app/
├── main.py
└── routers/
    └── resources.py
```

In `resources.py`:

```
from fastapi import APIRouter

router = APIRouter(
    prefix="/api/resources",
    tags=["Resources"]
)


@router.get("/")
def get_resources():
    return []
```

Then in `main.py`:

```
from fastapi import FastAPI

from app.routers.resources import router as resources_router

app = FastAPI()

app.include_router(resources_router)
```

Now resource routes are separated from application setup.

---

36. Add a Service Layer

Eventually, avoid putting business logic inside the router.

Instead of:

```
@router.post("/")
def create_resource(resource: ResourceCreate):
    # validation
    # business rules
    # database logic
    # calculations
    # notifications
    ...
```

use:

```
@router.post("/")
def create_resource(resource: ResourceCreate):
    return resource_service.create(resource)
```

Then:

```
Router
   ↓
Service
   ↓
Repository
   ↓
Database
```

This separation becomes valuable as the application grows.

---

37. Repository Layer

The repository handles data access.

For example:

```
class ResourceRepository:

    def get_all(self):
        ...

    def get_by_id(self, resource_id):
        ...

    def create(self, resource):
        ...

    def update(self, resource_id, data):
        ...

    def delete(self, resource_id):
        ...
```

Then the service doesn't need to know exactly how the database works.

```
Service
   ↓
Repository
   ↓
PostgreSQL
```

This makes it easier to change the data layer later.

---

38. Database Architecture

Our final architecture might look like:

```
                  Client
                    ↓
                 FastAPI
                    ↓
                 Router
                    ↓
               Service
                    ↓
             Repository
                    ↓
                ORM
                    ↓
             PostgreSQL
```

And the response:

```
PostgreSQL
    ↓
   ORM
    ↓
Repository
    ↓
 Service
    ↓
 Router
    ↓
Pydantic Response Model
    ↓
   JSON
```

This is much closer to a production backend.

---

39. Authentication

A real API often needs authentication.

For example:

```
POST /api/auth/login
```

could authenticate the user.

Then:

```
GET /api/resources
Authorization: Bearer <token>
```

could access protected resources.

Remember:

```
Authentication
    ↓
Who are you?

Authorization
    ↓
What are you allowed to do?
```

These are different concepts.

FastAPI's dependency system can be used to implement reusable authentication and authorization checks.

---

40. Authorization Example

Imagine:

```
User
Admin
```

A normal user can:

```
GET resources
```

but only an admin can:

```
DELETE resources
```

The flow becomes:

```
DELETE /api/resources/5
        ↓
Authenticate user
        ↓
Identify user
        ↓
```

```
Check permission
        ↓
Allowed?
   ↙         ↘
 Yes          No
 ↓             ↓
Delete        403
```

This is an authorization problem, not merely authentication.

---

41. CORS

Suppose:

```
Frontend
https://haas.dev

API
https://api.example.com
```

The browser may enforce cross-origin rules.

FastAPI can use CORS middleware to define which origins are allowed.

For development, you might allow a local frontend.

For production, configure only the origins that actually need access.

Avoid treating:

```
allow_origins=["*"]
```

as a universal production solution.

---

42. Testing the REST API

A REST API should be tested automatically.

Typical tests:

```
GET /resources
GET /resources/{id}
POST /resources
PUT /resources/{id}
PATCH /resources/{id}
DELETE /resources/{id}
```

Also test failure cases:

```
404
400
422
401
403
```

Example:

```
def test_get_resources(client):
    response = client.get("/api/resources")

    assert response.status_code == 200
```

Another:

```
def test_resource_not_found(client):
    response = client.get("/api/resources/9999")

    assert response.status_code == 404
```

Testing should verify both the status code and the response data.

---

43. Testing POST

Example:

```
def test_create_resource(client):

    response = client.post(
        "/api/resources",
        json={
            "title": "FastAPI Testing",
            "description": "Testing APIs",
            "category": "Python",
            "level": "Intermediate"
        }
    )

    assert response.status_code == 201

    data = response.json()

    assert data["title"] == "FastAPI Testing"
```

This verifies:

```
Request
 ↓
Validation
 ↓
Creation
 ↓
Response
```

---

44. Test Invalid Input

Send:

```
response = client.post(
    "/api/resources",
    json={
        "title": "FastAPI"
    }
)
```

Required fields are missing.

The test should verify that the API rejects the request rather than creating an incomplete resource.

This is an important distinction:

```
Successful request testing
+
Failure testing
```

Both are necessary.

---

45. API Documentation as a Development Tool

Open:

```
/docs
```

You should see:

```
Resources

GET    /api/resources
POST   /api/resources
GET    /api/resources/{resource_id}
PUT    /api/resources/{resource_id}
```

```
PATCH  /api/resources/{resource_id}
DELETE /api/resources/{resource_id}
```

You can select an endpoint and click:

```
Try it out
```

Then enter request data and execute it.

This makes `/docs` useful not just for consumers, but also during development and debugging.

---

46. Complete Small REST API

A compact version of our API:

```
from fastapi import FastAPI, HTTPException, status
from pydantic import BaseModel


app = FastAPI(
    title="Resource REST API",
    version="1.0.0"
)


class ResourceCreate(BaseModel):
    title: str
    description: str
    category: str
    level: str


class ResourceUpdate(BaseModel):
    title: str | None = None
    description: str | None = None
    category: str | None = None
    level: str | None = None


class ResourceResponse(BaseModel):
    id: int
    title: str
    description: str
    category: str
    level: str
```

```
resources = [
    {
        "id": 1,
        "title": "Python Basics",
        "description": "Learn Python fundamentals.",
        "category": "Python",
        "level": "Beginner"
    },
    {
        "id": 2,
        "title": "FastAPI Fundamentals",
        "description": "Learn FastAPI fundamentals.",
        "category": "Python",
        "level": "Intermediate"
    }
]


@app.get(
    "/api/resources",
    response_model=list[ResourceResponse]
)
def get_resources(
    category: str | None = None,
    level: str | None = None,
    search: str | None = None,
    skip: int = 0,
    limit: int = 20
):

    result = resources

    if category:
        result = [
            resource
            for resource in result
            if resource["category"].lower()
            == category.lower()
        ]
```

```
    if level:
        result = [
            resource
            for resource in result
            if resource["level"].lower()
            == level.lower()
        ]

    if search:
        search = search.lower()

        result = [
            resource
            for resource in result
            if search in resource["title"].lower()
            or search in resource["description"].lower()
        ]

    return result[skip:skip + limit]


@app.get(
    "/api/resources/{resource_id}",
    response_model=ResourceResponse
)
def get_resource(resource_id: int):

    for resource in resources:
        if resource["id"] == resource_id:
            return resource

    raise HTTPException(
        status_code=404,
        detail="Resource not found"
    )


@app.post(
    "/api/resources",
    response_model=ResourceResponse,
    status_code=status.HTTP_201_CREATED
)
def create_resource(resource: ResourceCreate):
```

```
    new_resource = {
        "id": len(resources) + 1,
        **resource.model_dump()
    }

    resources.append(new_resource)

    return new_resource


@app.put(
    "/api/resources/{resource_id}",
    response_model=ResourceResponse
)
def update_resource(
    resource_id: int,
    resource: ResourceCreate
):

    for existing in resources:

        if existing["id"] == resource_id:
            existing.update(
                resource.model_dump()
            )

            return existing

    raise HTTPException(
        status_code=404,
        detail="Resource not found"
    )


@app.patch(
    "/api/resources/{resource_id}",
    response_model=ResourceResponse
)
def patch_resource(
    resource_id: int,
    resource: ResourceUpdate
):
```

```
    for existing in resources:

        if existing["id"] == resource_id:

            update_data = resource.model_dump(
                exclude_unset=True
            )

            existing.update(update_data)

            return existing

    raise HTTPException(
        status_code=404,
        detail="Resource not found"
    )


@app.delete(
    "/api/resources/{resource_id}",
    status_code=status.HTTP_204_NO_CONTENT
)
def delete_resource(resource_id: int):

    for index, resource in enumerate(resources):

        if resource["id"] == resource_id:
            resources.pop(index)
            return

    raise HTTPException(
        status_code=404,
        detail="Resource not found"
    )
```

This gives you a working REST API with:

```
CRUD
Validation
Path parameters
Query parameters
Filtering
Searching
```

```
Pagination
Response models
Status codes
Error handling
Automatic documentation
```

---

47. How the Complete Request Flows

Consider:

```
POST /api/resources
```

with:

```
{
    "title": "REST APIs",
    "description": "Learn REST API design.",
    "category": "Backend",
    "level": "Intermediate"
}
```

The complete flow is:

```
Client
  ↓
HTTP POST
  ↓
FastAPI Router
  ↓
ResourceCreate
  ↓
Pydantic Validation
  ↓
Python Function
  ↓
Business Logic
  ↓
Data Store
  ↓
ResourceResponse
  ↓
JSON
```

```
  ↓
201 Created
```

This is the core mental model you should remember.

---

48. What Happens When Something Goes Wrong?

Suppose the client requests:

```
GET /api/resources/999
```

Flow:

```
Client
  ↓
FastAPI
  ↓
Route
  ↓
Search data
  ↓
Resource missing
  ↓
HTTPException
  ↓
404
  ↓
JSON error
```

The API does not simply crash.

It communicates the problem using an HTTP status code.

---

49. REST API Design Checklist

Before calling an API complete, check:

Resources

```
Are resources clearly defined?
```

URLs

```
Do URLs represent resources rather than actions?
```

Methods

```
Are HTTP methods used consistently?
```

Validation

```
Is incoming data validated?
```

Responses

```
Are response models defined?
```

Status codes

```
Are success and error codes appropriate?
```

Errors

```
Are error responses predictable?
```

Pagination

```
Can large collections be paginated?
```

Filtering

```
Can clients efficiently find relevant data?
```

Security

```
Are authentication and authorization handled?
```

Testing

```
Are success and failure cases tested?
```

Documentation

```
Can another developer understand the API?
```

---

50. Common Mistakes

Mistake 1: Action-based URLs

Avoid:

```
/createResource
/deleteResource
/getResources
```

Prefer:

```
POST /resources
DELETE /resources/1
GET /resources
```

Mistake 2: Returning `200` for every situation

A missing resource should not normally look like a successful request.

Use:

```
404
```

for not found.

Mistake 3: No request validation

Never trust client input.

Use Pydantic models at the API boundary.

Mistake 4: Exposing internal fields

Do not accidentally return:

```
password_hash
internal_notes
private_tokens
database metadata
```

Use response models to control the public representation.

Mistake 5: No pagination

Avoid returning enormous collections.

Mistake 6: One model for everything

Separate:

```
Create
Update
Response
Database
```

models when their responsibilities differ.

Mistake 7: Putting everything in the route

A route should not contain hundreds of lines of business logic.

Use:

```
Router
 ↓
Service
 ↓
Repository
```

Mistake 8: Treating the Python list as a production database

The list is only for learning.

Real applications need persistent storage.

---

51. Mini Project: haas.dev Resource API

Build a complete backend for learning resources.

Resource fields

```
id
title
description
category
level
slug
created_at
updated_at
```

Endpoints

```
GET    /api/v1/resources
GET    /api/v1/resources/{id}
POST   /api/v1/resources
PUT    /api/v1/resources/{id}
PATCH  /api/v1/resources/{id}
DELETE /api/v1/resources/{id}
```

Query features

```
?category=Python
```

```
?level=Beginner
```

```
?search=flask
```

```
?page=1&limit=20
```

Add

- Pydantic schemas
- Response models
- Validation
- Error handling
- Pagination
- Filtering
- Search
- API versioning
- Authentication
- Authorization
- Automated tests
- Database integration
- OpenAPI documentation

Suggested architecture

```
app/
│
├── main.py
│
├── routers/
│   └── resources.py
│
├── schemas/
│   └── resource.py
│
├── services/
│   └── resource_service.py
│
├── repositories/
│   └── resource_repository.py
│
├── models/
```

```
|   └── resource.py
|
└── dependencies.py
```

The goal is not simply to make endpoints.

The goal is to understand how a real backend is structured.

---

52. Five-Minute Challenge

Build:

```
GET /api/v1/health
```

Return:

```
{
    "status": "healthy",
    "service": "haas-resource-api",
    "version": "1.0.0"
}
```

Then build:

```
GET /api/v1/resources/{id}
```

Requirements:

- Integer ID
- Response model
- `404` when missing
- JSON response
- Visible in `/docs`

---

53. Exercises

Exercise 1: Add Slugs

Add:

```
slug
```

Example:

```
{
    "id": 1,
```

```
    "title": "FastAPI Fundamentals",
    "slug": "fastapi-fundamentals"
}
```

Then create:

```
GET /api/resources/fastapi-fundamentals
```

Exercise 2: Add Sorting

Support:

```
/api/resources?sort=title
```

and:

```
/api/resources?sort=created_at
```

Exercise 3: Add Pagination Metadata

Return:

```
{
    "data": [],
    "page": 1,
    "limit": 20,
    "total": 100
}
```

Exercise 4: Add Categories

Create:

```
GET /api/categories
```

Return unique categories.

Exercise 5: Add Authentication

Protect:

```
POST /api/resources
PUT /api/resources/{id}
PATCH /api/resources/{id}
DELETE /api/resources/{id}
```

while allowing public users to:

```
GET /api/resources
GET /api/resources/{id}
```

---

54. Interview Questions

Beginner

1. What is REST?
2. What is a REST API?
3. What is a resource?
4. What is the difference between an endpoint and a resource?
5. What does GET do?
6. What does POST do?
7. What does PUT do?
8. What does PATCH do?
9. What does DELETE do?
10. What does HTTP 404 mean?

Intermediate

11. What is the difference between PUT and PATCH?
12. Why should API URLs represent resources?
13. Why are Pydantic models useful in FastAPI?
14. What is a response model?
15. Why use different create and response models?
16. What is API versioning?
17. Why is pagination important?
18. What are query parameters used for?
19. How should an API handle missing resources?
20. Why should APIs use appropriate HTTP status codes?

Advanced

21. How would you structure a large FastAPI REST API?
22. What belongs in a router versus a service?
23. What is the repository pattern?
24. How would you implement authentication and authorization?
25. How would you protect sensitive response fields?
26. How would you design pagination for millions of database records?

27. How would you handle API versioning without breaking existing clients?
28. How would you test a REST API?
29. How would you design consistent error responses?
30. How would you migrate an in-memory API to PostgreSQL?

---

55. REST API Cheat Sheet

GET collection

```
@app.get("/resources")
def get_resources():
    ...
```

GET individual resource

```
@app.get("/resources/{resource_id}")
def get_resource(resource_id: int):
    ...
```

POST

```
@app.post(
    "/resources",
    status_code=201
)
def create_resource(resource: ResourceCreate):
    ...
```

PUT

```
@app.put("/resources/{resource_id}")
def update_resource(
    resource_id: int,
    resource: ResourceCreate
):
    ...
```

PATCH

```
@app.patch("/resources/{resource_id}")
def patch_resource(
    resource_id: int,
    resource: ResourceUpdate
```

```
):
    ...
```

DELETE

```
@app.delete(
    "/resources/{resource_id}",
    status_code=204
)
def delete_resource(resource_id: int):
    ...
```

Query parameters

```
@app.get("/resources")
def get_resources(
    category: str | None = None,
    limit: int = 20
):
    ...
```

Request model

```
class ResourceCreate(BaseModel):
    title: str
    category: str
```

Response model

```
class ResourceResponse(BaseModel):
    id: int
    title: str
    category: str
```

Error

```
raise HTTPException(
    status_code=404,
    detail="Resource not found"
)
```

Documentation

```
/docs
/redoc
```

---

56. Key Takeaways

- REST APIs organize application functionality around resources.
- URLs identify resources.
- HTTP methods communicate the intended operation.
- `GET` reads.
- `POST` creates.
- `PUT` updates or replaces.
- `PATCH` partially updates.
- `DELETE` removes.
- FastAPI uses Python type hints and Pydantic models to define API contracts.
- Request models validate incoming data.
- Response models control outgoing data.
- HTTP status codes communicate the result of an operation.
- `404` represents a missing resource.
- `201` is commonly used after successful creation.
- `204` represents successful completion with no response body.
- Query parameters are useful for filtering, searching, sorting, and pagination.
- Pagination becomes important as collections grow.
- API versioning helps manage breaking changes.
- Authentication identifies the caller.
- Authorization determines what that caller is allowed to do.
- Routers, services, and repositories help separate responsibilities in larger applications.
- Automated tests should cover both successful and failed requests.
- Automatic OpenAPI documentation makes the API contract easier to inspect and consume.

Final Mental Model

```
                    CLIENT
                      |
                      ↓
                HTTP REQUEST
                      |
                      ↓
                FASTAPI ROUTER
```

```
                    |
                    ↓
                SERVICE
                    |
                    ↓
              REPOSITORY
                    |
                    ↓
                DATABASE
                    |
                    ↓
              RESPONSE MODEL
                    |
                    ↓
              JSON RESPONSE
                    |
                    ↓
                 CLIENT
```

The important progression is:

```
FastAPI
   ↓
Endpoints
   ↓
REST resources
   ↓
CRUD
   ↓
Validation
   ↓
Status codes
   ↓
Response models
   ↓
Filtering + pagination
   ↓
Routers + services
   ↓
Database
   ↓
Authentication + authorization
```

```
    ↓
  Testing
    ↓
  Production API
```

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app